1 SUPPLEMENTARY MATERIAL

1.1 Algorithm: from alignments to compressed de Bruijn grpah

In this section we present an algorithm that constructs a compressed de Bruijn graph from the set of self-alignments of length $\geq k$ in the genome. We use mummer (Kurtz *et al.*, 2004) to preprocess the genome and efficiently locate all self-alignments in the genome whose lengths are at least k. Our alignment-based algorithm begins with a graph consisting of one large node to represent the entire genome. Then, the algorithm considers one alignment at a time. As each alignment is incorporated into the graph, the nodes are split to represent smaller subsequences of the genome. Occasionally, nodes are merged when a repetition is detected in the genome. Thus our algorithm achieves a runtime that is related to the number of self-alignments bounded by k. The number of self-alignments shrinks rapidly as k grows. In contrast there is no such advantage to using large values of k in an uncompressed de Bruijn graph because the initial number of nodes is fixed by the genome size.

Algorithm 2 depicts our alignment-based algorithm for constructing the compressed de Bruijn graph that represents a genome. We exclude implementation details that ensure the correctness of the algorithm. Each node captures a distinct subsequence of length $\geq k$ in the genome. This is stored as a set of start positions and a length. We maintain separately a sorted set of all start positions in the graph, with pointers to the nodes that represent them, so that we can quickly navigate to a start position in the graph and easily query whether there is a node with a particular start position. Each distinct subsequence of length $\geq k$ in the genome is represented by exactly one node in the compressed de Bruijn graph. Each k-mer, which we denote by its start position in the genome, is included in exactly one node in the graph. This invariant is true in the final graph as well as during construction. In the final graph, there is one leaf, representing the end of the genome. However, during construction, we allow many nodes to be leaves, representing suffixes of the genome. At the end of construction, each leaf (except possibly the shortest one) has its sequence truncated and becomes a parent to the first node whose sequence begins within the leaf's sequence.

We now summarize the procedure of our algorithm as it processes an alignment. We first insert the starting position of the first interval in the alignment, alignBeg1, to the appropriate node and then we add the starting position of the second interval, alignBeg2, to the same node. If another node already represents alignBeg2, the nodes are merged. Before merging nodes that begin with identical subsequences, we ensure that they represent sequences of the same length and precede a merge by a split (splitBackwards) if one node's sequence is a proper prefix of the other's. When a starting position is added to a node, we ensure that the subsequence is removed from any other node that already captured it, splitting nodes as appropriate. Thus we ensure that there are no redundancies in the graph.

When an alignment is considered, there are several scenarios that can occur when we insert alignBeg1.

1. **align1.beg is a starting position of a node in the graph.**

   a. **The existing node represents a longer subsequence than the alignment.** In this case, we split the existing node to form a new node whose sequence is a proper prefix of the existing node's. This uses the splitBackwards routine. Then a new start position of align2.beg is inserted to the new node [if it was not already there].

   b. **The existing node represents a shorter subsequence than the alignment.** In this case, we insert the beginning of the alignment by inserting a new start position of align2.beg to the new node. Then the alignment is trimmed at its left end and we continue by iterating through the rest of the alignment.

   c. **The existing node represents precisely the first interval of the alignment.** In this case, align2.beg is added as a start position for the node.

2. **align1.beg is not a starting position of any node in the graph.** In other words, alignBeg1 is implicitly included within a node.

   a. **The closest existing node with a start position before align1.beg ends at align1.end** In this case, we use splitForwards to split the closest node with a start position less than align1.beg into two nodes. Then align2.beg is inserted as a start position to the node that represents a suffix of the original node.

   b. **The closest existing node with a start position before align1.beg extends past align1.end** In this case, we use splitMiddle to split the closest node with a start position less than align1.beg. This creates two new nodes. align2.beg is inserted as a start position to the new node that represents the middle of the original node.

As the alignments are considered, nodes in the graph are merged and split. There are three ways in which a node is split, which we call splitBackwards, splitForwards, and splitMiddle. The splitBackwards routine is used when an alignment is a prefix to an existing node. It splits the existing node into two nodes. The splitForwards routine is used when an alignment is implicitly contained within an existing node, is not a prefix, and the alignment is a suffix of the existing node. It splits the existing node into two nodes. The splitMiddle routine is used when an alignment is implicitly contained within an existing node, is not a prefix, and the alignment ends earlier in the sequence than the existing node. It splits the existing node into three nodes. The splitting routines are depicted in Figure 1.

Self-overlapping alignments contributed additional complexity to our algorithm. Self-overlapping alignments are tandem repeats in the genome. We break down each self-overlapping alignment into its smallest repeating unit and create a node to capture the tandem repeat with all of its start positions. Then we create a separate node that bridges the occurrences of the tandem repeats, forming a cycle in the graph. We create an edge between these two nodes with multipflicity to represent all recurrences of the tandem repeat.

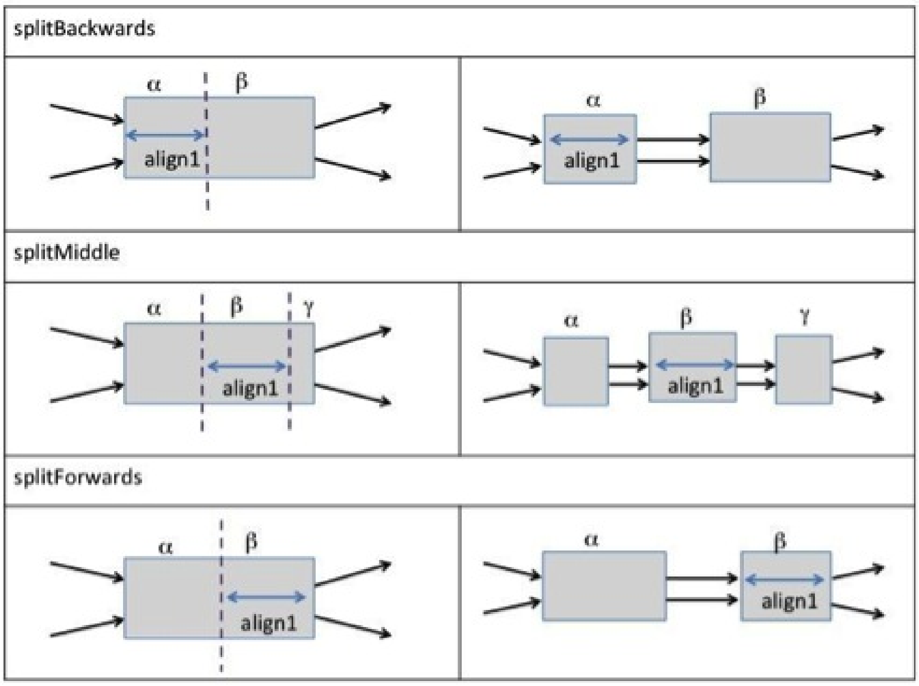

Fig. 1. The three splitting routines in our alignment-based algorithm. splitBackwards splits a node representing $\alpha\beta$ into separate nodes for α and β. splitMiddle splits a node representing $\alpha\beta\gamma$ into separate nodes for α, β and γ. splitForwards splits a node representing $\alpha\beta$ into separate nodes for α and β. Note that when a node representing the subsequence $\alpha\beta$ is split into separate nodes for α and β, the overlapping $k-1$ characters occur both at the end of the node for α and at the beginning of the node for β.

Algorithm 2 Construct Compressed de Bruijn Graph from Alignments

```
Input: genome sequence, $k$, set of self-alignments $\geq k$.
Output: compressed forward de Bruijn graph of genome.
for all lines in mummerOutputFile do
    if splitInterval then
                                        ▷ set align1 and align2 to second part of self-ovlerap
        splitInterval ← false
    else
                                        ▷ load align1 and align2 from input file
        if self overlapping alignment then
                                        ▷ split alignment to two parts
                                        ▷ set align1 and align2 to first part of self-overlap
            splitInterval ← true
        else
            splitInterval ← false
        end if
    end if
    while ! intervalInserted do
        foundPos = findNodeBeginAtPos(align1.beg)
        if foundPos $\not\equiv$ -1 then
            foundNode ← nodes[foundPos]
            if foundNode.length $>$ alignLength then
                                        ▷ foundNode is too long
                splitBackwards(foundNode, alignLength)
                intervalInserted ← true
            else if foundNode.length $<$ alignLength then
                                        ▷ foundNode is too short
                incToNextBegin ← foundNode.length $-k$ +1
                align1.beg+= incToNextBegin
                align2.beg+= incToNextBegin
                intervalInserted ← false
            else
                                        ▷ first interval is represented by foundNode
                intervalInserted ← true
            end if
        else
                                        ▷ align1.beg not found, implicitly included in a node
            lastNode ← closest node with start before align1.beg
            if align1.end is end of lastNode then
                foundNode ← splitNodeForward(lastNode, align1.beg)
            else
                foundNode ← splitNodeMiddle(lastNode, align1.beg, align1.length)
            end if
                                        ▷ foundNode represents align.beg
            createChild(newNode, align.beg)
            intervalInserted ← true
        end if
        addedStart ← addStartPosToNode(foundNode, align2.beg)
        if intervalInserted and addedStart then
            createChild(foundNode, align2.beg)
        end if
    end while
end for
updateLeaves()
```

Algorithm 3 Construct Repeat Nodes from MEM nodes in suffix tree in $O(n\ log\ n)$ time and space

```
procedure CREATEREPEATNODESFROMSUFFIXTREE                          ▷ recursive DFS of suffix tree
    CREATEREPEATNODESFROMMEM(root)
end procedure

procedure CREATEREPEATNODESFROMMEM(node)
    for all node children do
        CREATEREPEATNODESFROMMEM(node.child)
    end for
    if node.MEM then
        if node.parent $\neq$ root then                            ▷ include path from root to MEM node
            extend node label left to include path label from root
        end if
        while node.strdepth $\geq k$ do
            LMAnode $\leftarrow$ node.LMA
            if LMAnode $\neq$ null then                           ▷ skip LMAnode.strdepth characters
                if skippedChars then
                    createRepeatNode for skipped segment of MEM
                end if
                numCharsToSkip $\leftarrow$ LMAnode.strdepth $-k+1$
            end if
            node $\leftarrow$ node.suffixSkips[0]
            if numCharsToSkip $> 0$ then      ▷ use suffix skips to traverse numCharsToSkip suffix links quickly
                numCharsToSkip$--$
                if node.MEM then
                    break
                end if
                while numCharsToSkip $> 0$ do
                    slinkIndex $\leftarrow$ floor(log(numCharsToSkip) / log(2))
                    slinkTraversing $\leftarrow$ pow(2, slinkIndex)
                    if node.closestLMA[slinkIndex] $\neq$ null then
                        if node.closestLMAproximity[slinkIndex] $<$ numCharsToSkip then
                            adjust numCharsToSkip to extend over skipped LMA
                        end if
                    end if
                    node $\leftarrow$ node.suffixSkips[slinkIndex]
                    numCharsToSkip $-=$ slinkTraversing
                end while
            end if
        end while
        if needLastNode then
            createRepeatNode for overhang beyond last embedded MEM
        end if
    end if
end procedure
```

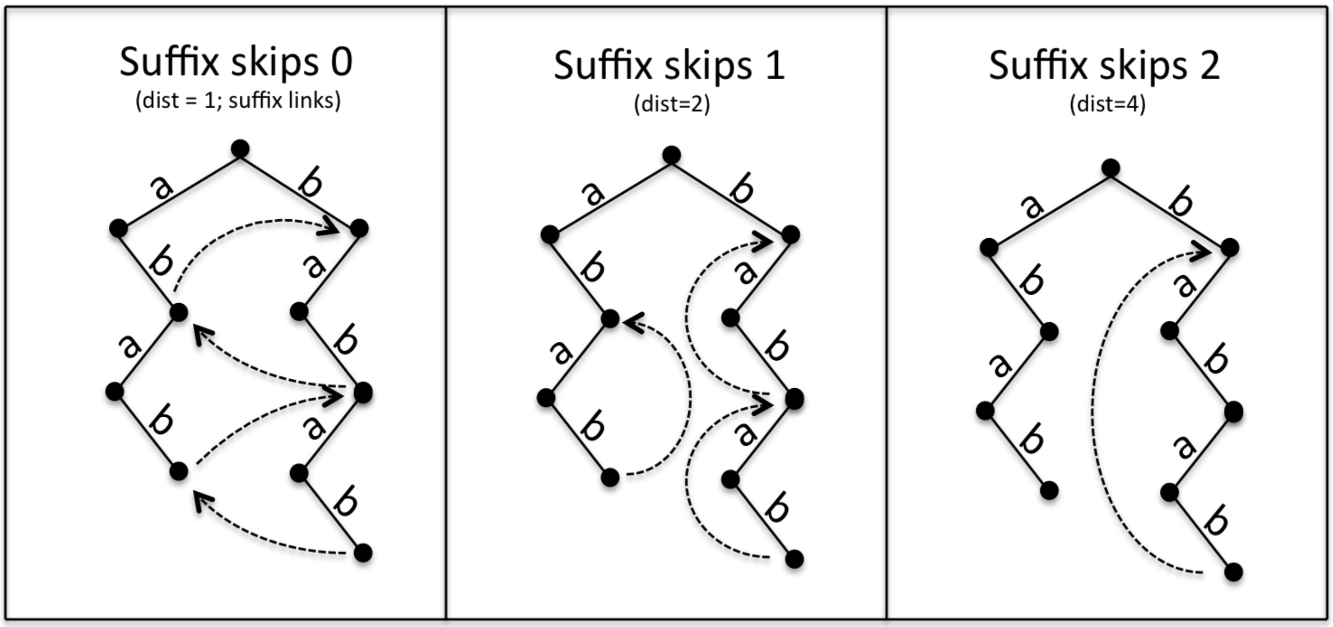

Fig. 2. Example suffix tree and suffix skips for the string "babab$". For clarity, only a subset of the suffix links and skips are displayed. Leaf nodes with $ characters are also not shown.

Table 1. The 9 *E. coli* and 9 *B. anthracis* strains included in our pan-genome analysis.

Strain	Size	Accession
B. anthracis A0248 uid33543	5178 KB	CP001598
B. anthracis A16R uid40353	5179 KB	CP001974
B. anthracis A16 uid40303	5179 KB	CP001970
B. anthracis Ames 0581 uid10784	5178 KB	AE017334
B. anthracis Ames uid309	5178 KB	AE016879
B. anthracis CDC 684 uid31329	5181 KB	CP001215
B. anthracis CI uid36309	5147 KB	CP001746
B. anthracis H9401 uid49361	5170 KB	CP002091
B. anthracis str Sterne uid10878	5180 KB	AE017225
E. coli 0127 H6 E2348 69 uid32571	4919 KB	FM180568
E. coli 042 uid40647	5193 KB	FN554766
E. coli 536 uid16235	4893 KB	CP000247
E. coli 55989 uid33413	5107 KB	CU928145
E. coli ABU 83972 uid38725	5083 KB	CP001671
E. coli APEC O1 uid16718	5034 KB	CP000468
E. coli APEC O78 uid184588	4753 KB	CP004009
E. coli BL21 DE3 uid20713	4516 KB	CP001509
E. coli BL21 DE3 uid28965	4516 KB	AM946981

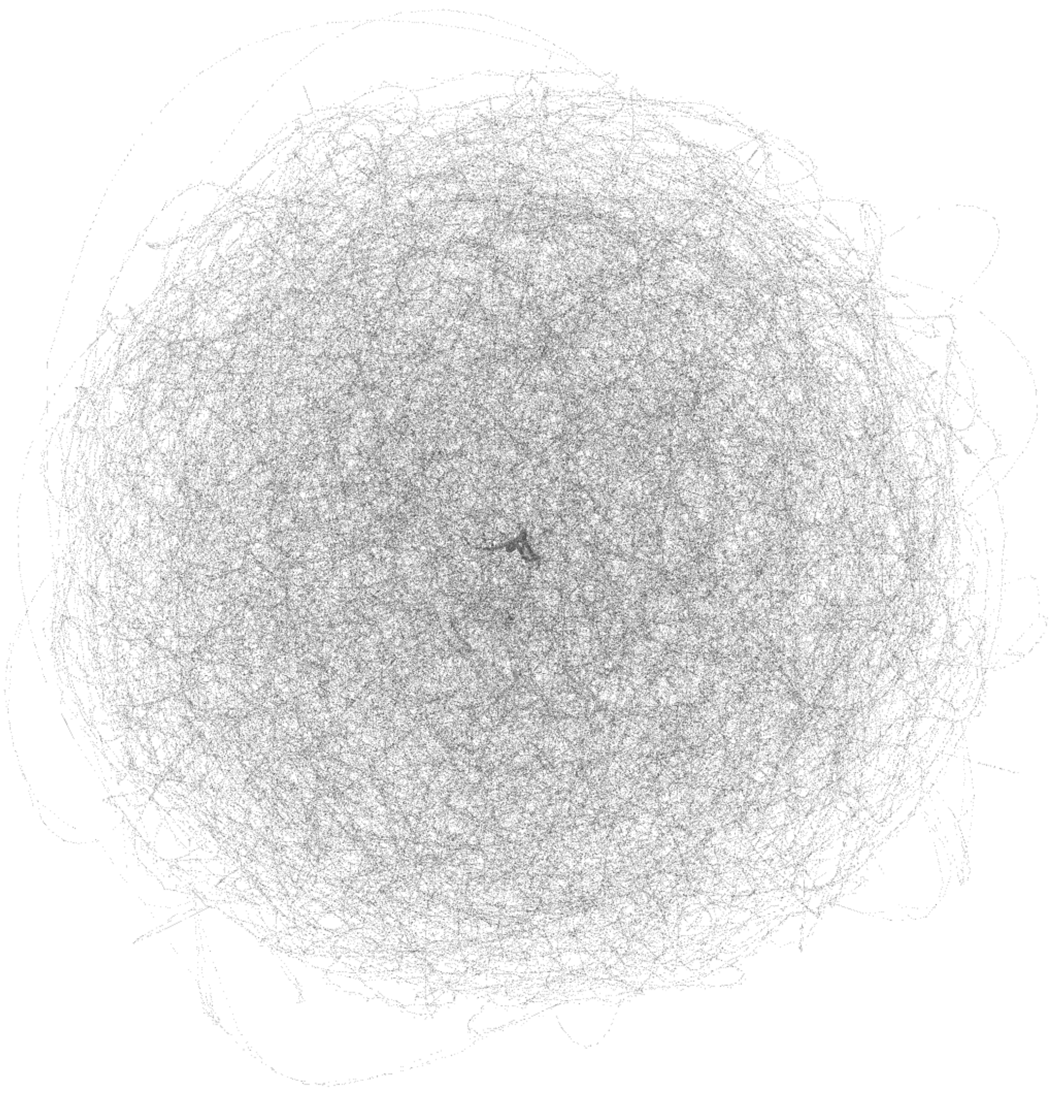

Fig. 3. The compressed de Bruijn graph for the B. anthracis pan genome with k=25 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

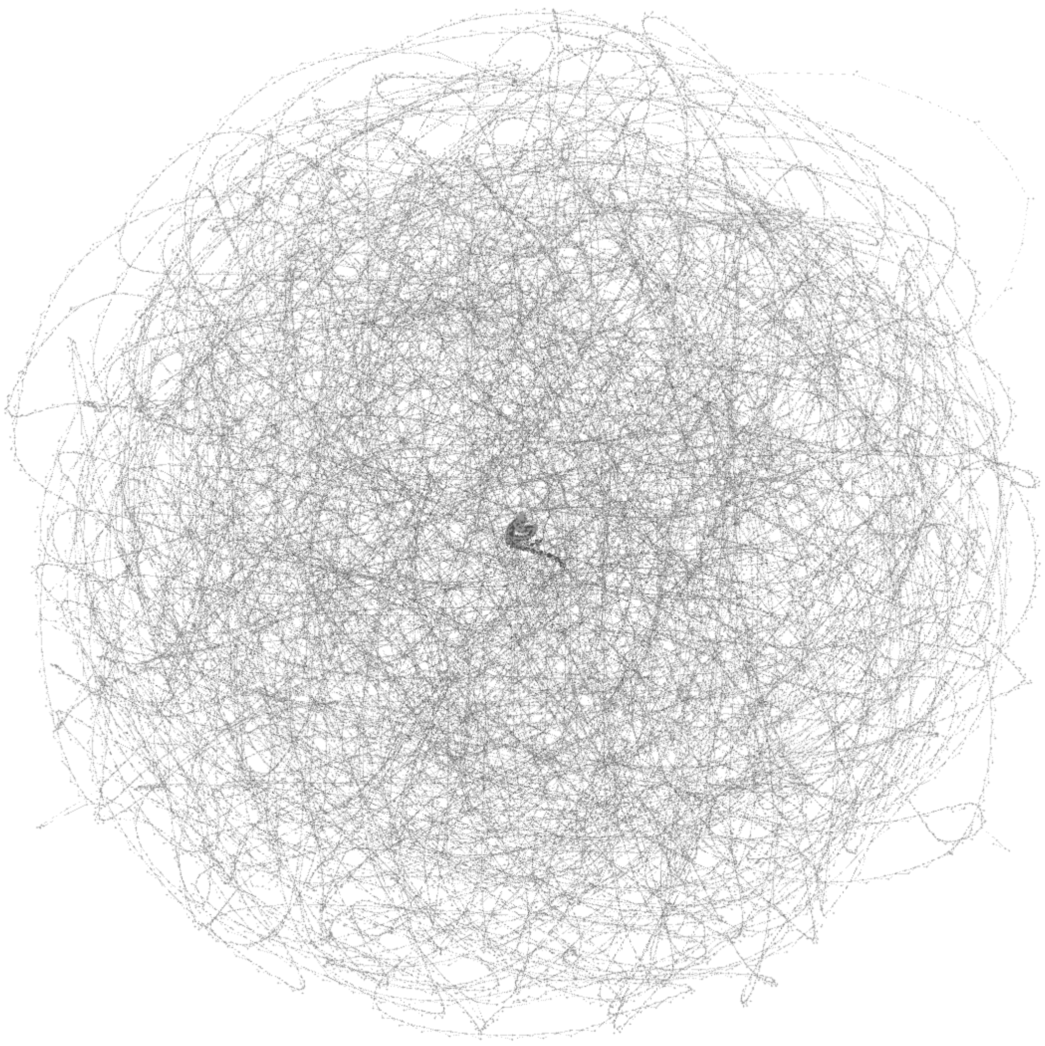

Fig. 4. The compressed de Bruijn graph for the B. anthracis pan genome with k=100 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

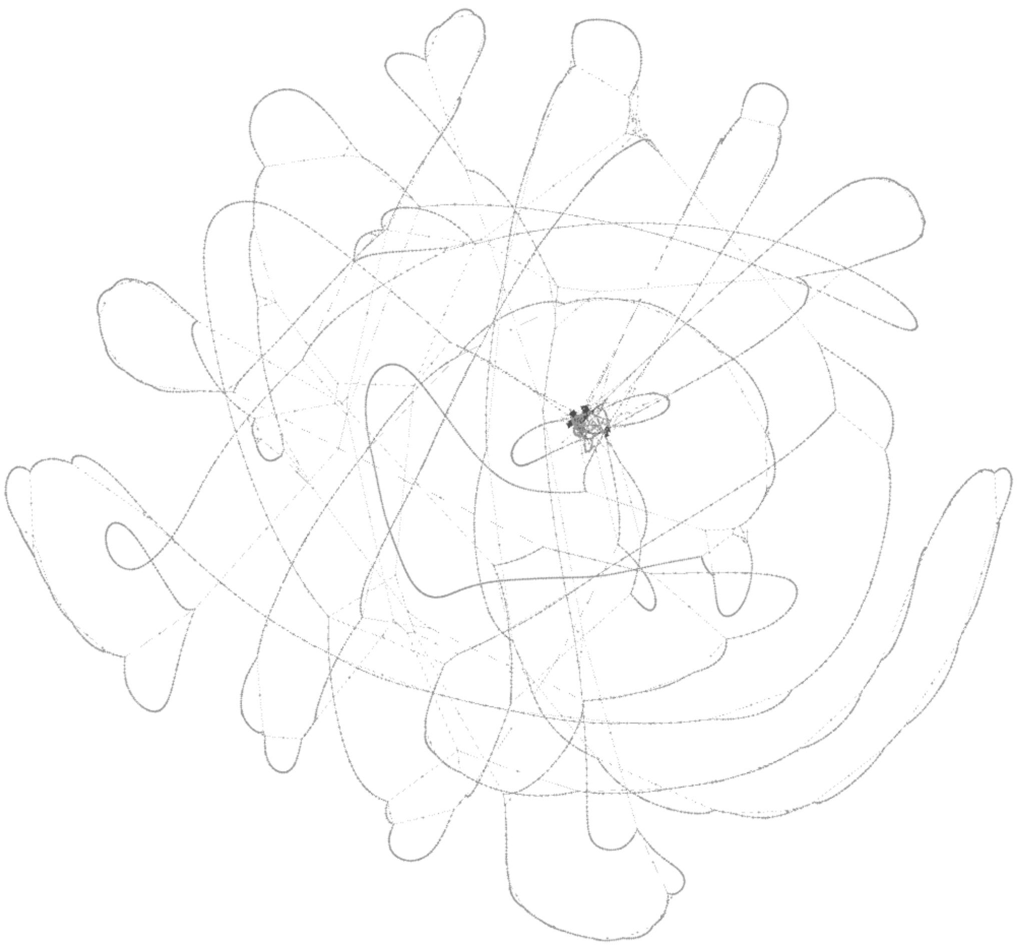

Fig. 5. The compressed de Bruijn graph for the B. anthracis pan genome with k=1000 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

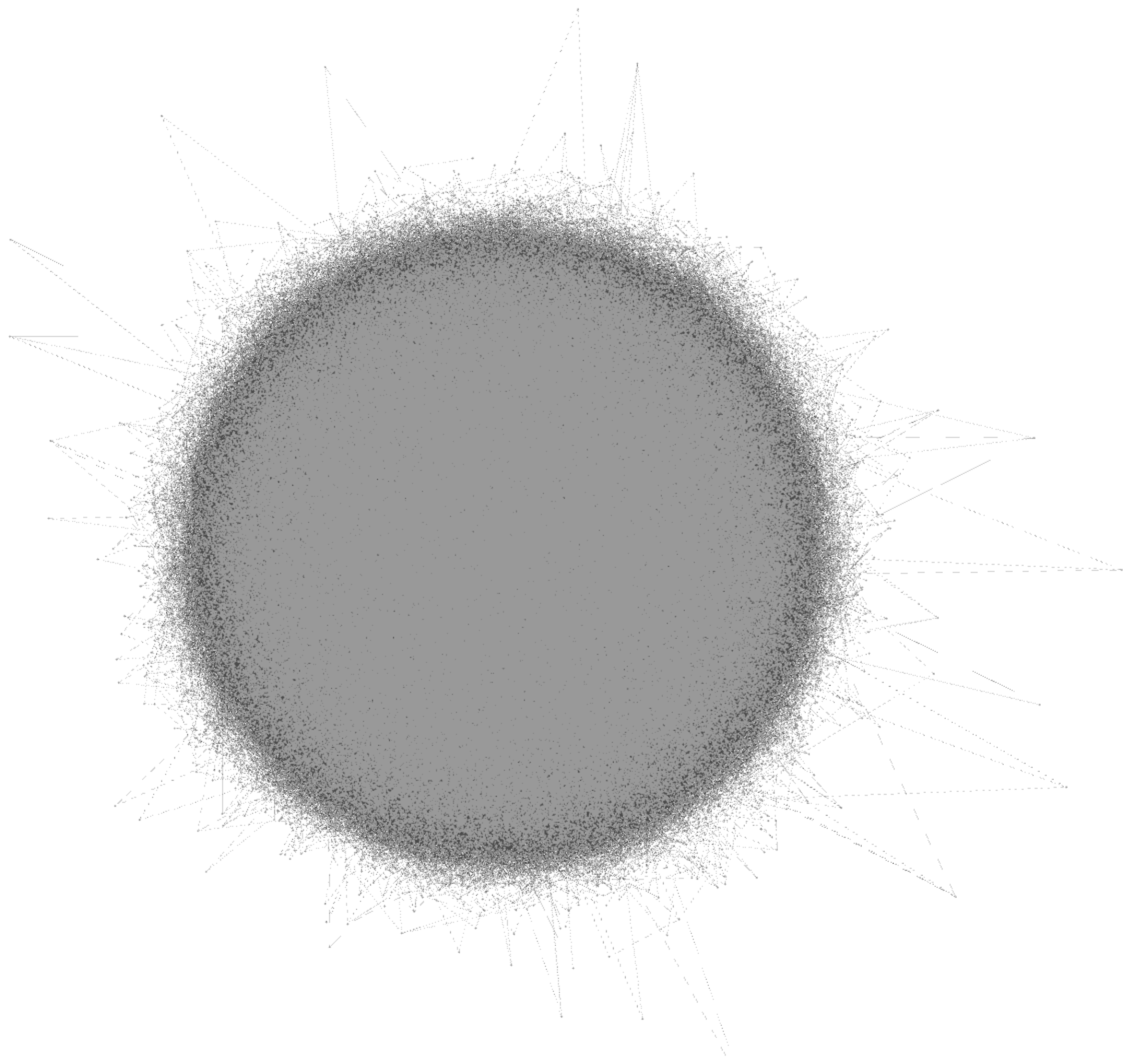

Fig. 6. The compressed de Bruijn graph for the E. coli pan genome with k=25 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

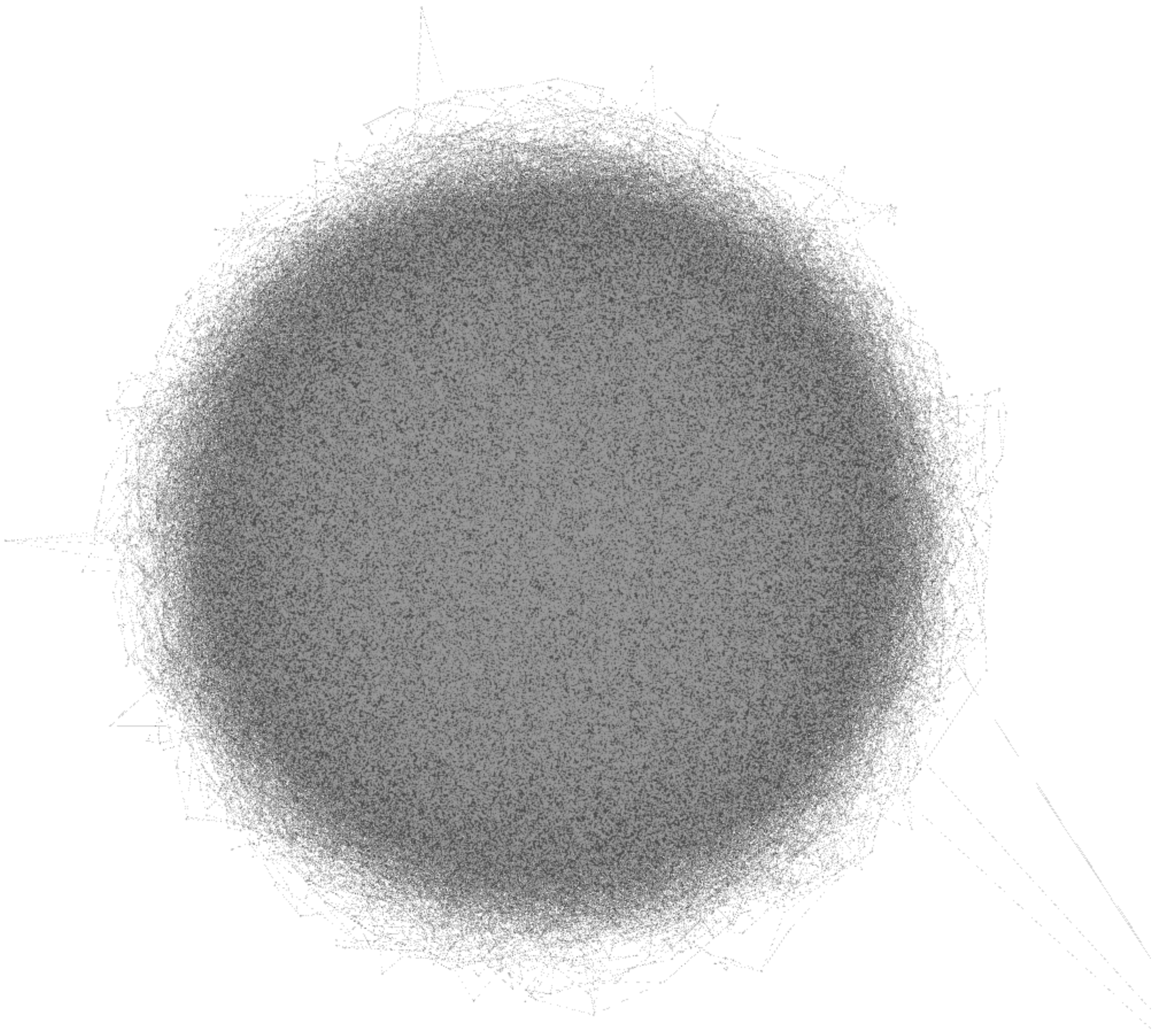

Fig. 7. The compressed de Bruijn graph for the E. coli pan genome with k=100 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

Fig. 8. The compressed de Bruijn graph for the E. coli pan genome with k=1000 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

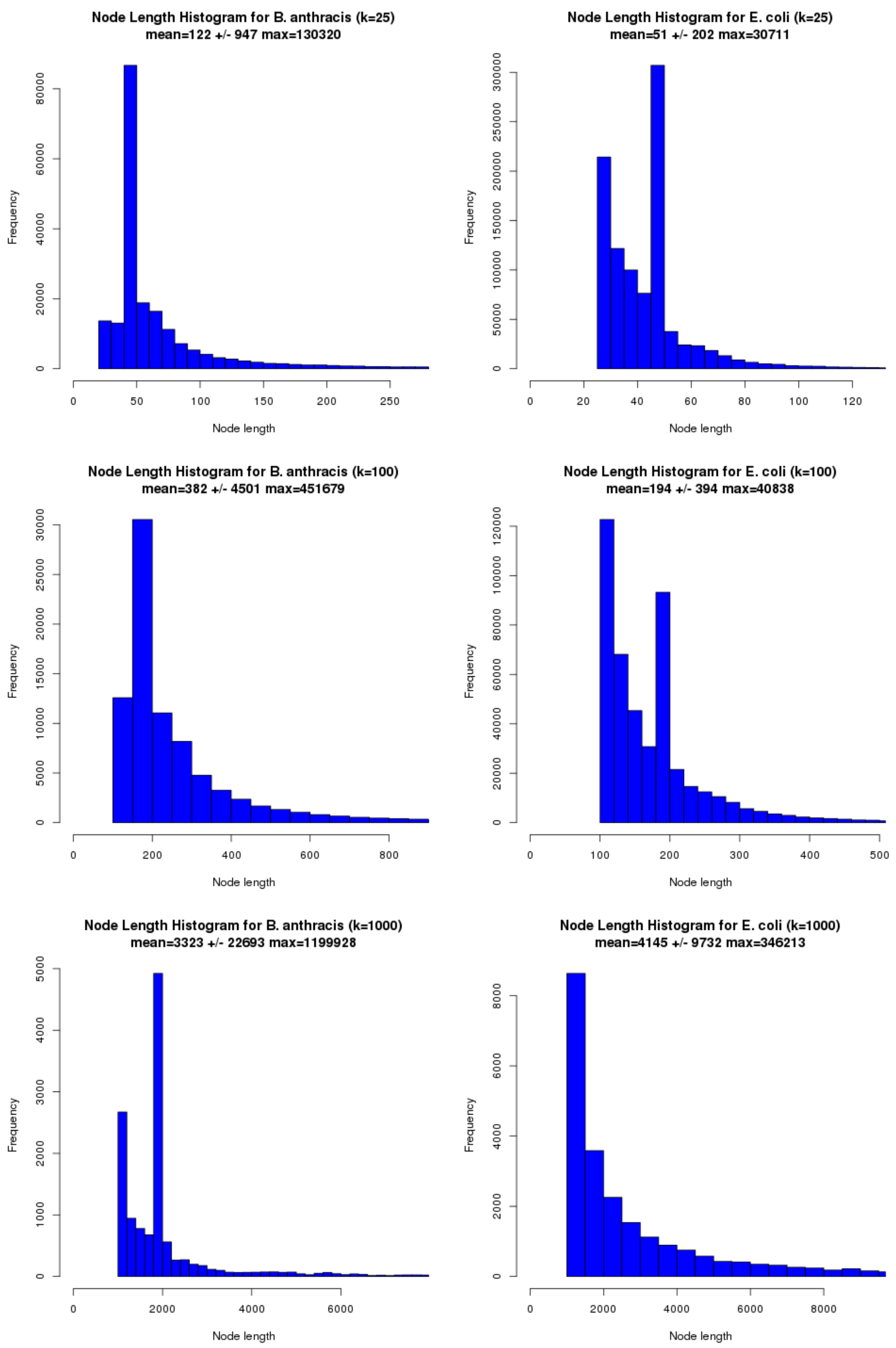

Fig. 9. Distributions of node lengths in the compressed de Bruijn graphs for the pan-genomes of 9 strains of *E. coli* and 9 strains of *B. anthracis.*

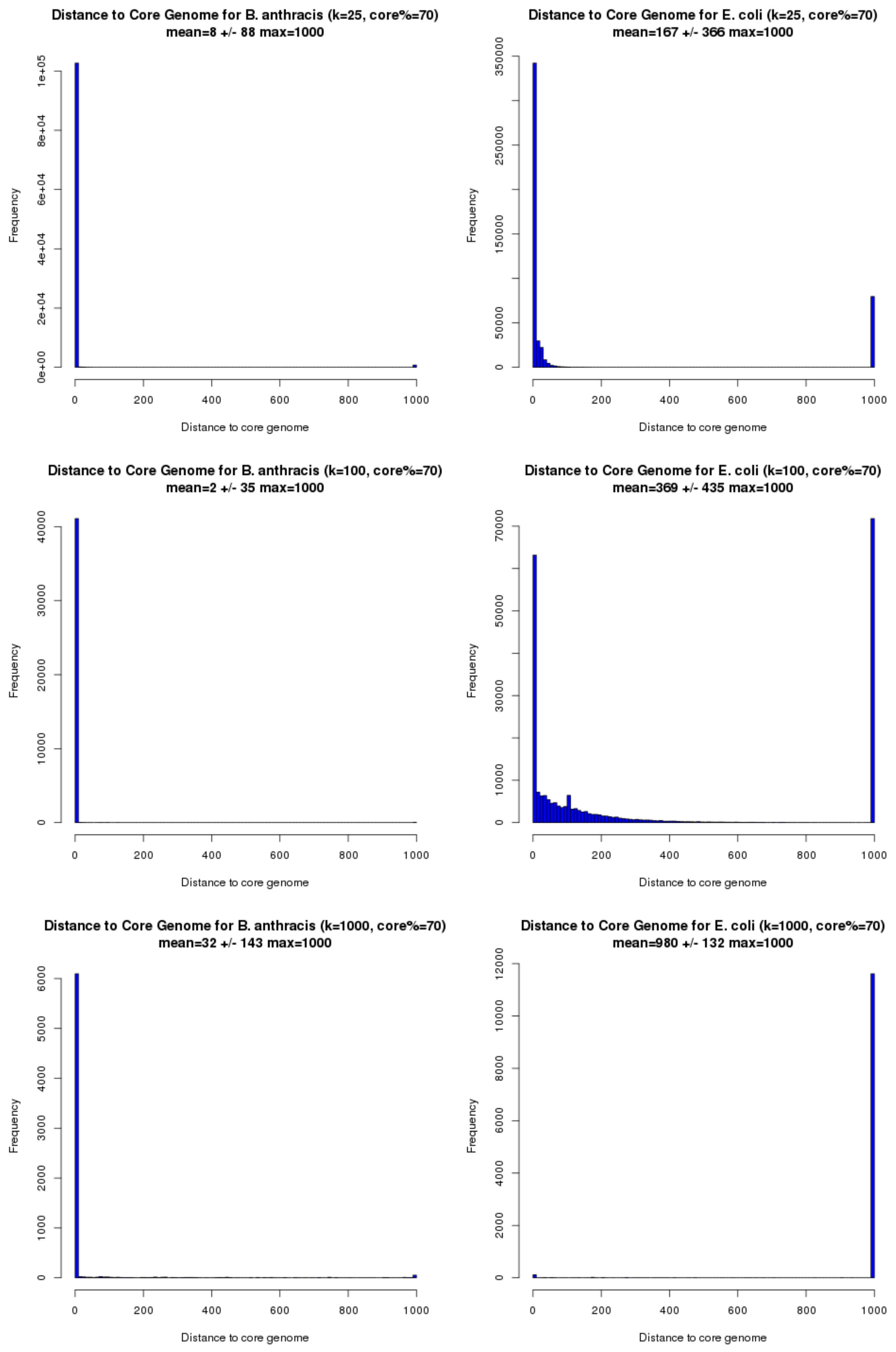

Fig. 10. Distributions of distances to the core genome in the compressed de Bruijn graphs for the pan-genomes of 9 strains of *E. coli* and 9 strains of *B. anthracis.*